CS 380S

TOCTTOU Attacks

Don Porter

Some slides courtesy Vitaly Shmatikov and Emmett Witchel

Definitions

- TOCTTOU – Time of Check To Time of Use
- Check – Establish some precondition (invariant), e.g., access permission
- Use – Operate on the object assuming that the invariant is still valid

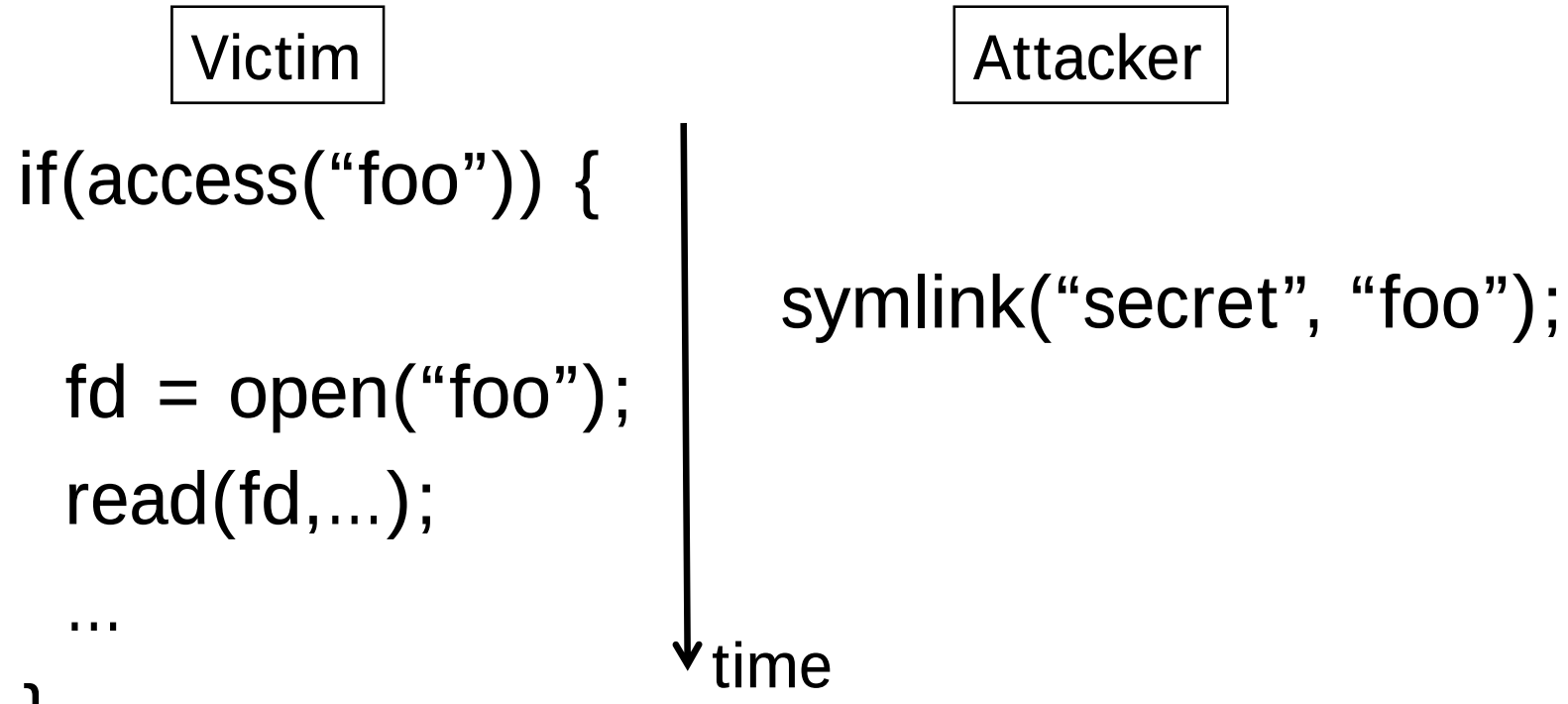
- Essentially a race condition
- Most famously in the file system, but can occur in any concurrent system

UNIX File System Security

- Access control: user should only be able to access a file if he has the permission to do so
- But what if user is running as setuid-root?
  - E.g., a printing program is usually setuid-root in order to access the printer device
    - Runs “as if” the user had root privileges
  - But a root user can access any file!
  - How does the printing program know that the user has the right to read (and print) any given file?
- UNIX has a special access() system call

TOCTTOU Example – setuid

- Victim checks file, if its good, opens it
- Attacker changes interpretation of file name
- Victim reads secret file



access()/open() Exploit

- Goal: trick setuid-root program into opening a normally inaccessible file
- Create a symbolic link to a harmless user file
  - access() will say that file is Ok to read
- After access(), but before open() switch symbolic link to point to /etc/shadow
  - /etc/shadow is a root-readable password file
- Attack program must run concurrently with the victim and switch the link at exactly the right time
  - Interrupt victim between access() and open()
  - How easy is this in practice?

Broken passwd

[Bishop]

- Password update program on HP/UX and SunOS (circa 1996)
- When invoked with password file as argument...
  1. Open password file and read the entry for the invoking user
  2. Create and open temporary file called ptmp in the same directory as password file
  3. Open password file again, update contents and copy into ptmp
  4. Close both password file and ptmp, rename ptmp to be the password file

TOCTTOU Attack on passwd

Create our own subdirectory FakePwd and fake password file pwdfile with blank root password; create symbolic link lnk->FakePwd; run passwd on lnk/pwdfile

1. Open password file and read the entry for the invoking user

   Change lnk->RealPwd to point to real password directory

2. Create and open temporary file called ptmp in the same directory as password file

   ptmp is created in RealPwd

   Change lnk->FakePwd to point to fake password directory

3. Open password file again, update contents and copy into ptmp

   contents read from FakePwd/pwdfile and copied to RealPwd/ptmp

   Change lnk->RealPwd to point to real password directory

4. Close both password file and ptmp, rename ptmp to password file

   Now RealPwd/pwdfile contains blank root password. Success!

Directory Removal Exploit

Recursive removal of a directory tree (GNU file utilities)

Original tree is /tmp/dir1/dir2/dir3

chdir(“/tmp/dir1”)
chdir(“dir2”)
chdir(“dir3”)
unlink(“*”)
chdir(“..”)
rmdir(“dir3”)
unlink(“*”)
chdir(“..”)
rmdir(“dir2”)
unlink(“*”)
rmdir(“/tmp/dir1”)

Suppose attacker executes **“mv /tmp/dir1/dir2/dir3 /tmp”** right here

This call will delete the entire root directory!

Fix: verify that inode of the directory did not change before and after chdir()

Temporary File Exploit

```
// Check if file already exists
if (stat(fn,&sb)==0) {
   fd = open(fn, O_CREAT | O_RDWR, 0);
   if (fd<0) {
      err(1, fn);
   }
}
```

Suppose attacker creates a symbolic link with the same name as *fn pointing to an existing file

This will overwrite the file to which attacker's link points

Evading System Call Interposition

- TOCTTOU and race conditions can be used to evade system call interposition by sharing state
- Example: when two Linux threads share file system information, they share their root directories and current working directory
  - Thread A's current working directory is /tmp
  - Thread A calls open("shadow"); B calls chdir("/etc")
    - Both look harmless; system monitor permits both calls
  - open("shadow") executes with /etc as working directory
    - A's call now opens "/etc/shadow" – oops!
- Similar attacks on shared file descriptors, etc.

Non-Filesystem Race Conditions

- Sockets: create/connect races for local daemons
  - OpenSSH < 1.2.17
- Symbolic links for Unix sockets
  - Plash
- Signal handlers
  - See Zalewski – “Sending signals for Fun and Profit”

TOCTTOU Vulnerabilities in Red Hat 9

National Vulnerability Database currently has 600 entries for symlink attack

Application	TOCTTOU errors	Possible exploit
vi	**<open, chown>**	Changing the owner of /etc/passwd to an ordinary user
gedit	**<rename, chown>**	Changing the owner of /etc/passwd to an ordinary user
rpm	**<open, open>**	Running arbitrary command
emacs	**<open,chmod>**	Making /etc/shadow readable by an ordinary user

• Jinpeng Wei, Calton Pu. FAST'05

How Hard Is It to Win a Race?

- Idea: force victim program to perform an expensive I/O operation
  - While waiting for I/O to complete, victim will yield CPU to the concurrent attack program, giving it window of opportunity to switch the symlink, working dir, etc.
- How? Make sure that the file being accessed is not in the file system cache
  - Force victim to traverse very deep directory structures (see Borisov et al. paper for details)

Maze Attack

Replace `/tmp/foo -> bar` with:

```
/tmp/foo
-> 1/a/b/c/d/e/...
-> 2/a/b/c/d/e/...
...
-> k/a/b/c/d/e/...
-> bar
```

Maze Attack, cont.

```
/tmp/foo
-> 1/a/b/c/d/e/...
-> 2/a/b/c/d/e/...
...
-> k/a/b/c/d/e/...
-> bar
```

```
1a/a/b/c/d/e/...
->2a/a/b/c/d/e/...
...
->ka/a/b/c/d/e/...
-> secret
```

1) Pollute OS cache with unrelated garbage
2) Pick an arbitrary file in maze, poll atime
3) On update, replace maze

Maze Recap

[Borisov et al.]

- Attacker must track victim's progress
  - When to insert symlink?
- After access started:
  - Monitor access time on a single directory entry
- Before open:
  - Force disk reads during access

How hard to prevent TOCTTOU?

- No portable, deterministic solution with current POSIX filesystem API – Dean and Hu 2004
- Tactics:
  1. Static checks for dangerous pairs (compile time)
  2. Hacks to setuid programs (least privilege)
  3. Kernel detection and compensation (RaceGuard)
  4. User-mode dynamic detection
  5. Change the interface

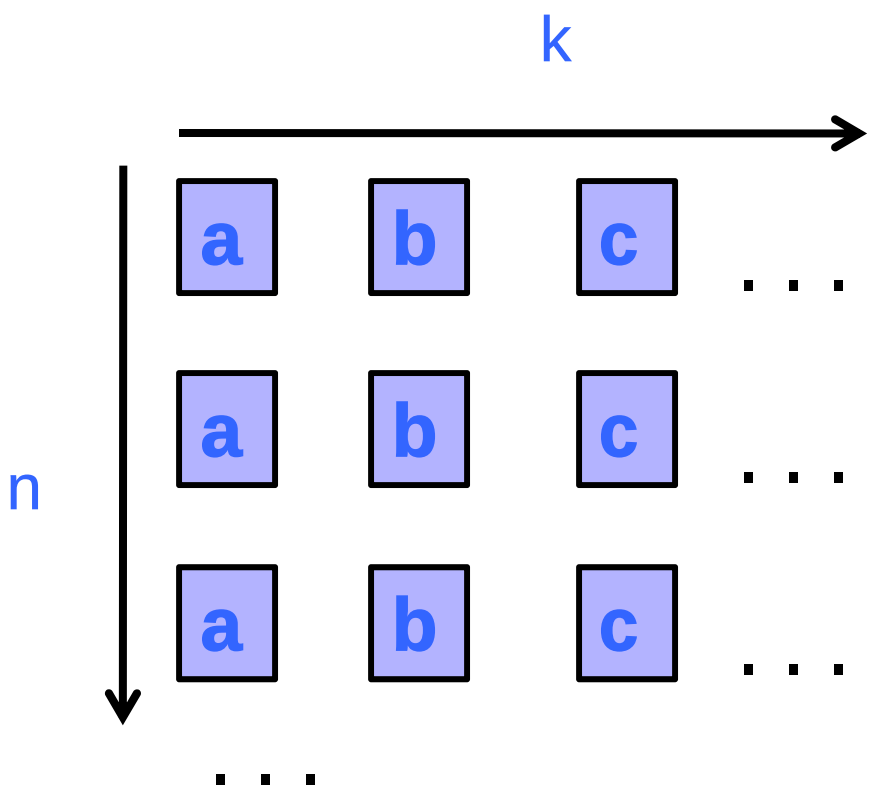
Hardness Amplification (Dean)

- If probability of attacker winning race is p<1,
- Essentially, do the access() *n* times and make sure they agree before doing the open()
- But what about mazes?
  - p == 1

Take 2 – (Tsafrir '08)

- Idea: Column-oriented traversal in userspace

  `/a/b/c/...`



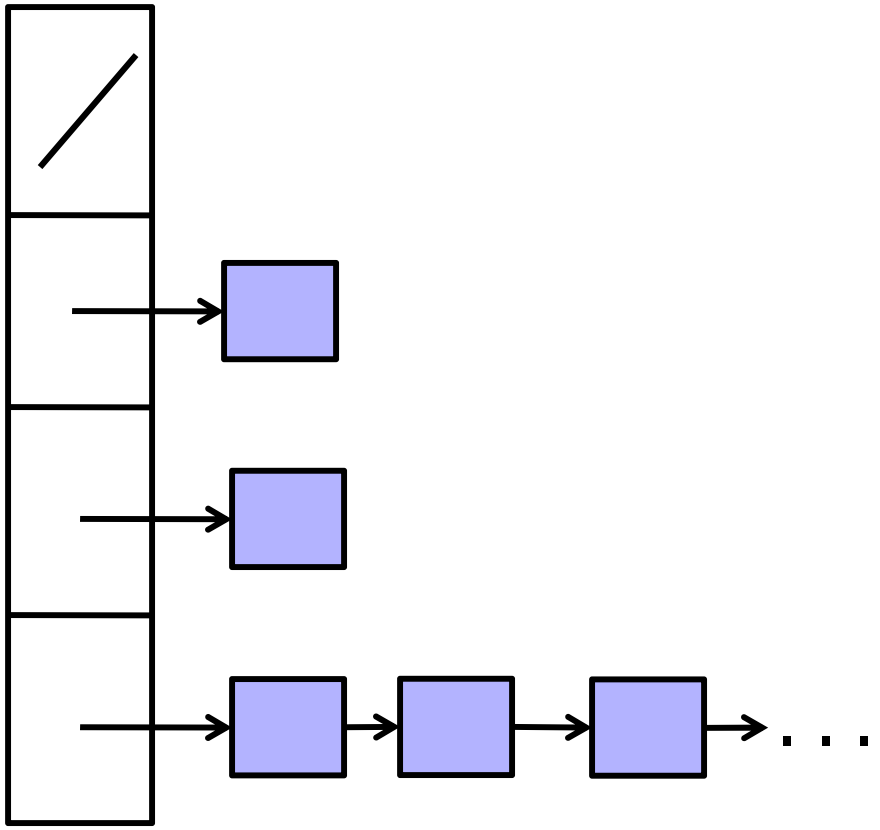
- Insight: hard to force scheduling in same directory
- Notes:
  - User space
  - Probabilistic

Cai et al. ‘09

- Idea: Algorithmic complexity attack on filesystem namespace
- Forced victim to be descheduled at end of each syscall without mazes
  - Even in same directory
- Paper also includes interesting scheduler priority manipulation

Linux dcache

- “foo” hashes to 3
- Pollute bucket 3 with garbage
- Victim burns timeslice traversing very long hash chain
- OS schedules attacker at end of syscall



Cai recap

- Disproved intuition about column traversal
- Generalization: probabilistic countermeasures unlikely to every work
  - Attackers likely to figure out how to single step victim
- Deterministic solutions are the only solutions

Tsafrir made Deterministic

- Insight 2: Hardness amplification not necessary
- Userspace traversal sufficient with *at() calls:

```
fd1 = open(“/”);
fstatat(fd1, &statbuf);
// do some checks
fd2 = openat(fd1, “a”);
fstatat(fd2, &statbuf);
// more checks
fd3 = openat(fd2, “b”);
...
```

Caveats

- Slower (many more syscalls)
- Incompatible with exec, O_CREAT
  - Re-opens door to temp file attacks
- Still requires API changes
  - openat(), fstatat(), etc.

How hard to prevent TOCTTOU?

- Tactics:
  1. Static checks for dangerous pairs (compile time)
     - Difficult in practice
  2. Hacks to setuid programs (least privilege)
     - Most common fix for single app
  3. ~~Kernel detection and compensation (RaceGuard)~~
  4. User-mode dynamic detection
     1. ~~Probabilistic~~
     2. Deterministic – Requires API Changes, Incomplete
  5. Change the interface
     - Most common approach to general problems

Adapting the API

- In the last 2 years, 13 new system calls have been added to Linux to prevent TOCTTOU
  - openat, renameat, etc. all take file descriptors
- In the last 3 years, new signal handling
  - pselect, ppoll change signal mask
- Current proposals for close-on-exec flag to the open system call
  - Prevents a race between open and fcntl (exploitable in a web browser)
- Cluttered and complicated APIs are the enemy of secure code

Transactions

- **A**tomic: either the entire transaction succeeds or fails
- **C**onsistent: transactions represent a consistent data structure update
- **I**solated: partial results are not visible to the rest of the system. This allows all transactions to be ordered (serialized).
- **D**urable: they survive computer failures
- Transactions help us reason about concurrency

Pseudo-Transactions

[Tsyrklevich and Yee]

- Observation: many sequences of filesystem operations are intended to be atomic
  - E.g., nothing should happen betw. access() and open()
- Pseudo-transaction: a sequence of filesystem calls that always behaves as if it were executed in isolation and free from interference
  - Very well-understood concept in databases
- Idea: OS should recognize when a file transaction starts and prevent interfering system calls

Tsyrklevich-Yee System

- Look at 2-call sequences of filesystem calls
  - Implemented as a kernel module
- Assume that first call starts a pseudo-transaction, second call ends it
  - Also need to time out misidentified transaction starts
- Treat all filesystem operations originating from the same process as part of same transaction
  - Assume process doesn't maliciously interfere with its own filesystem access
  - Assume fork()'d children run the same process image

...Also destroyed by Cai et al. ‘09

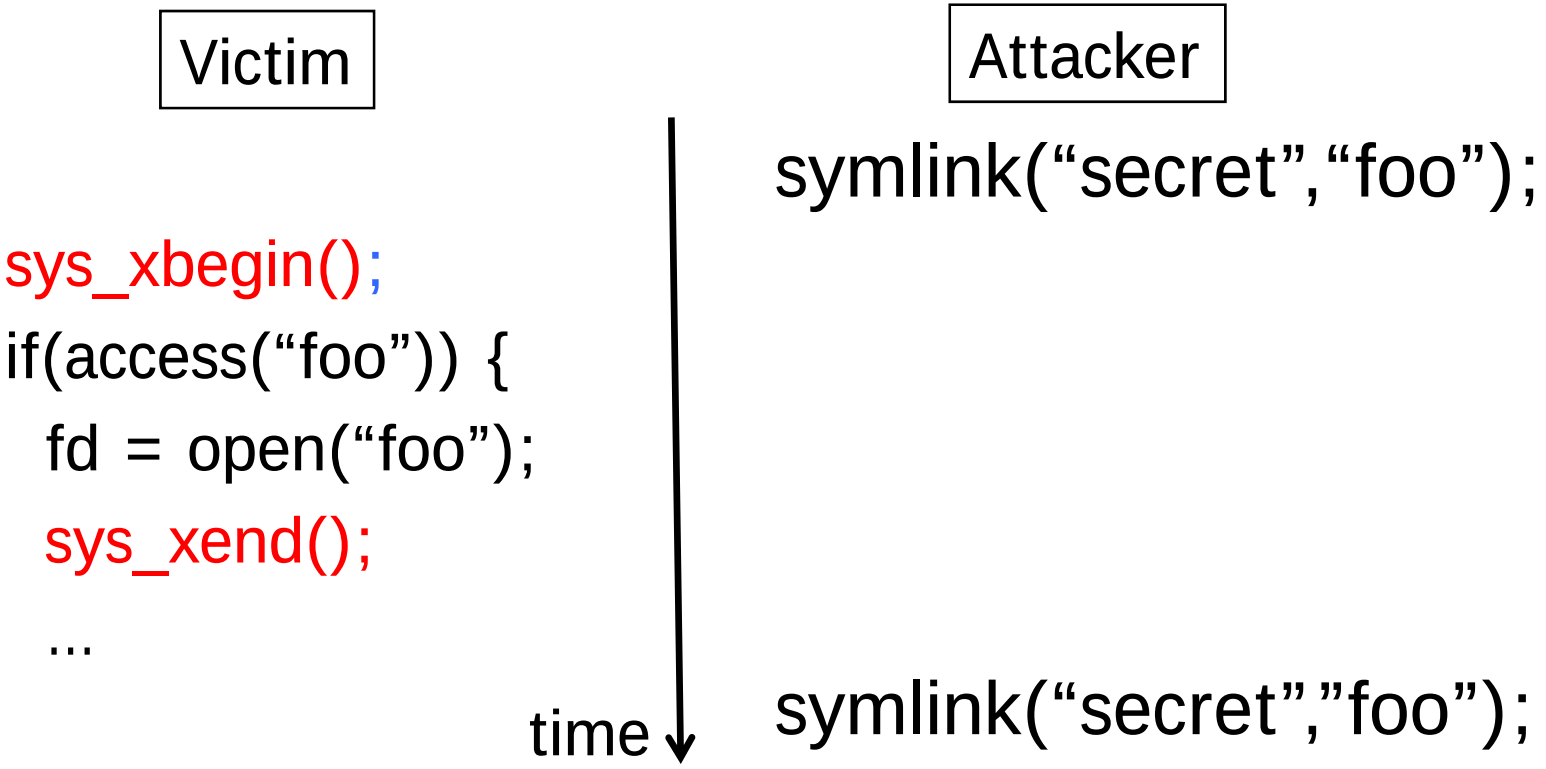
- Kernel has finite resources to track fs operations
- Idea: pollute the cache with enough garbage to evict first operation
  - Or manipulate scheduling for false timeout
- Varies by implementation

System Transactions – SOSP '09

- New system calls for transactions
  - sys_xbegin
  - sys_xend
  - sys_xabort
- System calls within an active transaction
  - atomic: all or nothing
  - isolated: partial results invisible
- Easy to adopt, just wrap code with transactions
- Deterministic guarantees

TOCTTOU Example Redux

- Attack ordered before or after check and use
  - System transactions save the day



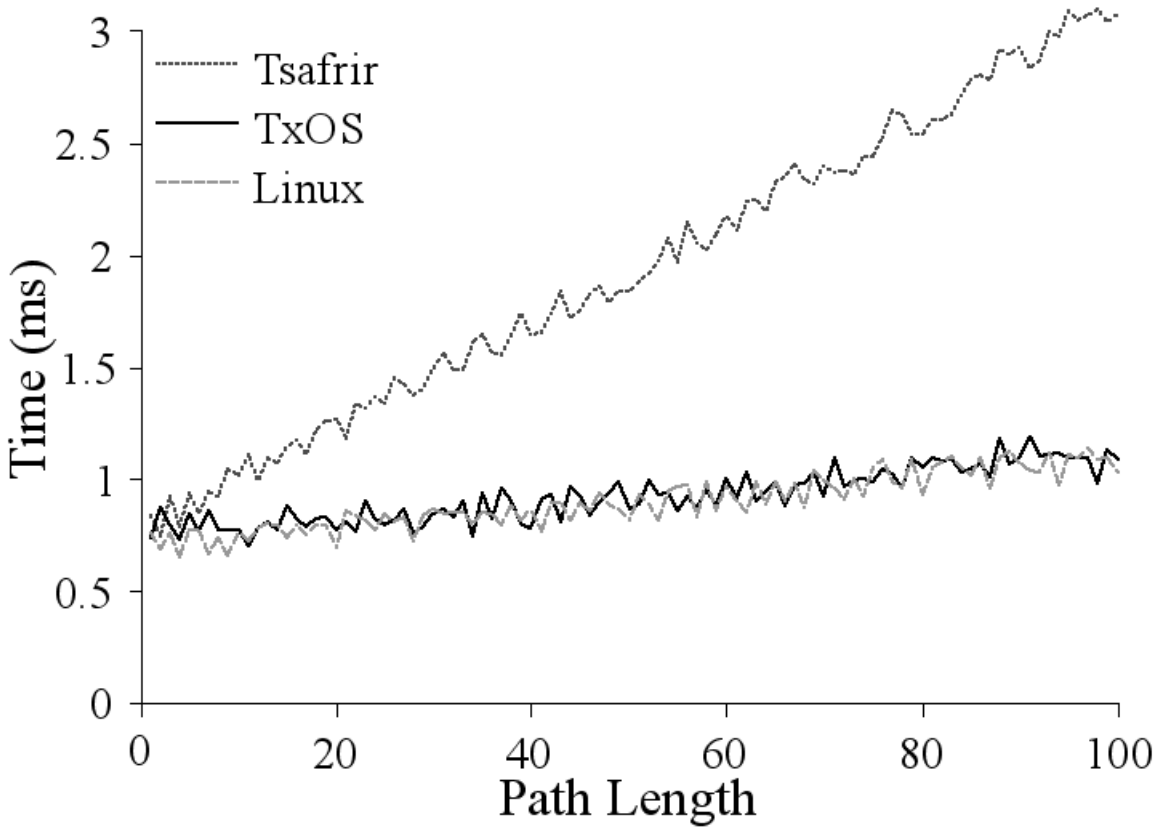
Prototype

- A version of Linux 2.6.22 modified to support system transactions
  - Affectionately called TxOS
  - Runs on commodity hardware
  - Supports a range of system calls
    - fs, memory allocation, fork, signals
- Reasonably efficient
  - Benchmark overheads: 1-2x
  - Some speedups!

Questions?

- porterde@cs

Preventing TOCTTOU Races



Typical Setuid-Root File Access

```
// Assume this is running inside some setuid-root program
void foo(char *filename) {
   int fd;
   if (access(filename, R_OK) != 0)
      exit(1);
   fd=open(filename, O_RDONLY);
   ... do something with fd ...
}
```

Check if user has the permission to read this file

What if the file to which filename points changed **right here**?

Open file for reading

This is known as a TOCTTOU attack

("Time of Check To Time of Use")

Fixing Race Conditions

- Unsafe sequence has been detected. What now?
- Roll back to state before transaction
  - Requires a heavy-duty file system
- Lock out other processes when a “critical section” of filesystem operations is being executed
  - How to identify critical sections?
  - One process gets a lock on entire filesystem (bad idea)
- “Delay-lock”: temporarily delay other processes trying to access a locked file
  - How to calculate the right delay? What if attacker wakes up before victim completes his file operation?

Default Allow Policy

- Allow every 2-call sequence except these:

ACCESS REMOVE

CHDIR REMOVE

EXEC REMOVE

where REMOVE = UNLINK | RMDIR | RENAME

Default Deny Policy

- Deny any 2-call sequence except these:

```
PERMIT(OPEN_RW,          OPEN_RW | ACCESS | UTIMES | CHDIR | EXEC |
                         UNLINK | READLINK | CHMOD | CHOWN | RENAME)
PERMIT(OPEN_CREAT,       OPEN_RW | ACCESS | UTIMES | CHDIR | EXEC |
                         RENAME_FROM)
PERMIT(ACCESS,           OPEN_RW | ACCESS | UTIMES | CHDIR | EXEC)
PERMIT(EXEC,             OPEN_READ | EXEC)
PERMIT(CHDIR,            OPEN_READ | CHDIR | ACCESS | READLINK)
PERMIT(RENAME_FROM,      OPEN_RW | ACCESS | UNLINK | RENAME_FROM)
PERMIT(RENAME_TO,        OPEN_RW)
PERMIT(CHMOD | CHOWN,    OPEN_RW | ACCESS | CHMOD | CHOWN)
PERMIT(UTIMES,           OPEN_RW | ACCESS | CHMOD | CHOWN)
PERMIT(READLINK,         READLINK)
```